Network Working Group
Request for Comments: 1320
Obsoletes: RFC 1186

R. Rivest
MIT Laboratory for Computer Science
and RSA Data Security, Inc.

April 1992

The MD4 Message-Digest Algorithm

Status of thie Memo

This memo provides information for the Internet community. It does not specify an Internet standard. Distribution of this memo is unlimited.

Acknowlegements

We would like to thank Don Coppersmith, Burt Kaliski, Ralph Merkle, and Noam Nisan for numerous helpful comments and suggestions.

Table of Contents

1. Executive Summary

This document describes the MD4 message-digest algorithm [1]. The algorithm takes as input a message of arbitrary length and produces as output a 128-bit "fingerprint" or "message digest" of the input. It is conjectured that it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given prespecified target message digest. The MD4 algorithm is intended for digital signature applications, where a large file must be "compressed" in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem such as RSA.

The MD4 algorithm is designed to be quite fast on 32-bit machines. In addition, the MD4 algorithm does not require any large substitution tables; the algorithm can be coded quite compactly.

The MD4 algorithm is being placed in the public domain for review and possible adoption as a standard.

This document replaces the October 1990 RFC 1186 [2]. The main difference is that the reference implementation of MD4 in the appendix is more portable.

For OSI-based applications, MD4's object identifier is

```
md4 OBJECT IDENTIFIER ::=
  {iso(1) member-body(2) US(840) rsadsi(113549) digestAlgorithm(2) 4}
```

In the X.509 type AlgorithmIdentifier [3], the parameters for MD4 should have type NULL.

2. Terminology and Notation

In this document a "word" is a 32-bit quantity and a "byte" is an eight-bit quantity. A sequence of bits can be interpreted in a natural manner as a sequence of bytes, where each consecutive group of eight bits is interpreted as a byte with the high-order (most significant) bit of each byte listed first. Similarly, a sequence of bytes can be interpreted as a sequence of 32-bit words, where each consecutive group of four bytes is interpreted as a word with the low-order (least significant) byte given first.

Let x_i denote "x sub i". If the subscript is an expression, we surround it in braces, as in x_{i+1}. Similarly, we use ^ for superscripts (exponentiation), so that x^i denotes x to the i-th power.

Let the symbol "+" denote addition of words (i.e., modulo-2^32 addition). Let X <<< s denote the 32-bit value obtained by circularly shifting (rotating) X left by s bit positions. Let not(X) denote the bit-wise complement of X, and let X v Y denote the bit-wise OR of X and Y. Let X xor Y denote the bit-wise XOR of X and Y, and let XY denote the bit-wise AND of X and Y.

3. MD4 Algorithm Description

We begin by supposing that we have a b-bit message as input, and that we wish to find its message digest. Here b is an arbitrary nonnegative integer; b may be zero, it need not be a multiple of eight, and it may be arbitrarily large. We imagine the bits of the message written down as follows:

m_0 m_1 ... m_{b-1}

The following five steps are performed to compute the message digest of the message.

3.1 Step 1. Append Padding Bits

The message is "padded" (extended) so that its length (in bits) is congruent to 448, modulo 512. That is, the message is extended so that it is just 64 bits shy of being a multiple of 512 bits long. Padding is always performed, even if the length of the message is already congruent to 448, modulo 512.

Padding is performed as follows: a single "1" bit is appended to the message, and then "0" bits are appended so that the length in bits of the padded message becomes congruent to 448, modulo 512. In all, at least one bit and at most 512 bits are appended.

3.2 Step 2. Append Length

A 64-bit representation of b (the length of the message before the padding bits were added) is appended to the result of the previous step. In the unlikely event that b is greater than 2^64, then only the low-order 64 bits of b are used. (These bits are appended as two 32-bit words and appended low-order word first in accordance with the previous conventions.)

At this point the resulting message (after padding with bits and with b) has a length that is an exact multiple of 512 bits. Equivalently, this message has a length that is an exact multiple of 16 (32-bit) words. Let M[0 ... N-1] denote the words of the resulting message, where N is a multiple of 16.

3.3 Step 3. Initialize MD Buffer

A four-word buffer (A,B,C,D) is used to compute the message digest. Here each of A, B, C, D is a 32-bit register. These registers are initialized to the following values in hexadecimal, low-order bytes first):

```
word A: 01 23 45 67
word B: 89 ab cd ef
word C: fe dc ba 98
word D: 76 54 32 10
```

3.4 Step 4. Process Message in 16-Word Blocks

We first define three auxiliary functions that each take as input three 32-bit words and produce as output one 32-bit word.

```
F(X,Y,Z) = XY v not(X) Z
G(X,Y,Z) = XY v XZ v YZ
H(X,Y,Z) = X xor Y xor Z
```

In each bit position F acts as a conditional: if X then Y else Z. The function F could have been defined using + instead of v since XY and not(X)Z will never have "1" bits in the same bit position.) In each bit position G acts as a majority function: if at least two of X, Y, Z are on, then G has a "1" bit in that bit position, else G has a "0" bit. It is interesting to note that if the bits of X, Y, and Z are independent and unbiased, the each bit of f(X,Y,Z) will be independent and unbiased, and similarly each bit of g(X,Y,Z) will be independent and unbiased. The function H is the bit-wise XOR or parity" function; it has properties similar to those of F and G.

Do the following:

```
   Process each 16-word block. */
   For i = 0 to N/16-1 do

     /* Copy block i into X. */
     For j = 0 to 15 do
       Set X[j] to M[i*16+j].
     end /* of loop on j */

     /* Save A as AA, B as BB, C as CC, and D as DD. */
     AA = A
     BB = B
     CC = C
     DD = D

     /* Round 1. */
     /* Let [abcd k s] denote the operation
          a = (a + F(b,c,d) + X[k]) <<< s. */
     /* Do the following 16 operations. */
     [ABCD  0  3]  [DABC  1  7]  [CDAB  2 11]  [BCDA  3 19]
     [ABCD  4  3]  [DABC  5  7]  [CDAB  6 11]  [BCDA  7 19]
     [ABCD  8  3]  [DABC  9  7]  [CDAB 10 11]  [BCDA 11 19]
     [ABCD 12  3]  [DABC 13  7]  [CDAB 14 11]  [BCDA 15 19]

     /* Round 2. */
     /* Let [abcd k s] denote the operation
          a = (a + G(b,c,d) + X[k] + 5A827999) <<< s. */
```

```
        /* Do the following 16 operations. */
        [ABCD  0  3]  [DABC  4  5]  [CDAB  8  9]  [BCDA 12 13]
        [ABCD  1  3]  [DABC  5  5]  [CDAB  9  9]  [BCDA 13 13]
        [ABCD  2  3]  [DABC  6  5]  [CDAB 10  9]  [BCDA 14 13]
        [ABCD  3  3]  [DABC  7  5]  [CDAB 11  9]  [BCDA 15 13]

        /* Round 3. */
        /* Let [abcd k s] denote the operation
             a = (a + H(b,c,d) + X[k] + 6ED9EBA1) <<< s. */
        /* Do the following 16 operations. */
        [ABCD  0  3]  [DABC  8  9]  [CDAB  4 11]  [BCDA 12 15]
        [ABCD  2  3]  [DABC 10  9]  [CDAB  6 11]  [BCDA 14 15]
        [ABCD  1  3]  [DABC  9  9]  [CDAB  5 11]  [BCDA 13 15]
        [ABCD  3  3]  [DABC 11  9]  [CDAB  7 11]  [BCDA 15 15]

        /* Then perform the following additions. (That is, increment each
           of the four registers by the value it had before this block
           was started.) */
        A = A + AA
        B = B + BB
        C = C + CC
        D = D + DD

      end /* of loop on i */
```

Note. The value 5A..99 is a hexadecimal 32-bit constant, written with the high-order digit first. This constant represents the square root of 2. The octal value of this constant is 013240474631.

The value 6E..A1 is a hexadecimal 32-bit constant, written with the high-order digit first. This constant represents the square root of 3. The octal value of this constant is 015666365641.

See Knuth, The Art of Programming, Volume 2 (Seminumerical Algorithms), Second Edition (1981), Addison-Wesley. Table 2, page 660.

3.5 Step 5. Output

The message digest produced as output is A, B, C, D. That is, we begin with the low-order byte of A, and end with the high-order byte of D.

This completes the description of MD4. A reference implementation in C is given in the appendix.

4. Summary

The MD4 message-digest algorithm is simple to implement, and provides a "fingerprint" or message digest of a message of arbitrary length. It is conjectured that the difficulty of coming up with two messages having the same message digest is on the order of 2^64 operations, and that the difficulty of coming up with any message having a given message digest is on the order of 2^128 operations. The MD4 algorithm has been carefully scrutinized for weaknesses. It is, however, a relatively new algorithm and further security analysis is of course justified, as is the case with any new proposal of this sort.

References

[1] Rivest, R., "The MD4 message digest algorithm", in A.J. Menezes and S.A. Vanstone, editors, Advances in Cryptology - CRYPTO ’90 Proceedings, pages 303-311, Springer-Verlag, 1991.

[2] Rivest, R., "The MD4 Message Digest Algorithm", RFC 1186, MIT, October 1990.

[3] CCITT Recommendation X.509 (1988), "The Directory - Authentication Framework".

[4] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, MIT and RSA Data Security, Inc, April 1992.

APPENDIX A - Reference Implementation

This appendix contains the following files:

global.h -- global header file

md4.h -- header file for MD4

md4c.c -- source code for MD4

mddriver.c -- test driver for MD2, MD4 and MD5

The driver compiles for MD5 by default but can compile for MD2 or MD4 if the symbol MD is defined on the C compiler command line as 2 or 4.

The implementation is portable and should work on many different plaforms. However, it is not difficult to optimize the implementation on particular platforms, an exercise left to the reader. For example, on "little-endian" platforms where the lowest-addressed byte in a 32-bit word is the least significant and there are no alignment restrictions, the call to Decode in MD4Transform can be replaced with

a typecast.

A.1 global.h

```
/* GLOBAL.H - RSAREF types and constants
 */

/* PROTOTYPES should be set to one if and only if the compiler supports
     function argument prototyping.
   The following makes PROTOTYPES default to 0 if it has not already
     been defined with C compiler flags.
 */
#ifndef PROTOTYPES
#define PROTOTYPES 0
#endif

/* POINTER defines a generic pointer type */
typedef unsigned char *POINTER;

/* UINT2 defines a two byte word */
typedef unsigned short int UINT2;

/* UINT4 defines a four byte word */
typedef unsigned long int UINT4;

/* PROTO_LIST is defined depending on how PROTOTYPES is defined above.
   If using PROTOTYPES, then PROTO_LIST returns the list, otherwise it
     returns an empty list.
 */

#if PROTOTYPES
#define PROTO_LIST(list) list
#else
#define PROTO_LIST(list) ()
#endif
```

A.2 md4.h

```
/* MD4.H - header file for MD4C.C
 */

/* Copyright (C) 1991-2, RSA Data Security, Inc. Created 1991. All
   rights reserved.

   License to copy and use this software is granted provided that it
   is identified as the "RSA Data Security, Inc. MD4 Message-Digest
   Algorithm" in all material mentioning or referencing this software
   or this function.
```

```
   License is also granted to make and use derivative works provided
   that such works are identified as "derived from the RSA Data
   Security, Inc. MD4 Message-Digest Algorithm" in all material
   mentioning or referencing the derived work.

   RSA Data Security, Inc. makes no representations concerning either
   the merchantability of this software or the suitability of this
   software for any particular purpose. It is provided "as is"
   without express or implied warranty of any kind.

   These notices must be retained in any copies of any part of this
   documentation and/or software.
 */

/* MD4 context. */
typedef struct {
  UINT4 state[4];                                   /* state (ABCD) */
  UINT4 count[2];        /* number of bits, modulo 2^64 (lsb first) */
  unsigned char buffer[64];                         /* input buffer */
} MD4_CTX;

void MD4Init PROTO_LIST ((MD4_CTX *));
void MD4Update PROTO_LIST
  ((MD4_CTX *, unsigned char *, unsigned int));
void MD4Final PROTO_LIST ((unsigned char [16], MD4_CTX *));
```

A.3 md4c.c

```
/* MD4C.C - RSA Data Security, Inc., MD4 message-digest algorithm
 */

/* Copyright (C) 1990-2, RSA Data Security, Inc. All rights reserved.

   License to copy and use this software is granted provided that it
   is identified as the "RSA Data Security, Inc. MD4 Message-Digest
   Algorithm" in all material mentioning or referencing this software
   or this function.

   License is also granted to make and use derivative works provided
   that such works are identified as "derived from the RSA Data
   Security, Inc. MD4 Message-Digest Algorithm" in all material
   mentioning or referencing the derived work.

   RSA Data Security, Inc. makes no representations concerning either
   the merchantability of this software or the suitability of this
   software for any particular purpose. It is provided "as is"
   without express or implied warranty of any kind.
```

```
   These notices must be retained in any copies of any part of this
   documentation and/or software.
 */

#include "global.h"
#include "md4.h"

/* Constants for MD4Transform routine.
 */
#define S11 3
#define S12 7
#define S13 11
#define S14 19
#define S21 3
#define S22 5
#define S23 9
#define S24 13
#define S31 3
#define S32 9
#define S33 11
#define S34 15

static void MD4Transform PROTO_LIST ((UINT4 [4], unsigned char [64]));
static void Encode PROTO_LIST
  ((unsigned char *, UINT4 *, unsigned int));
static void Decode PROTO_LIST
  ((UINT4 *, unsigned char *, unsigned int));
static void MD4_memcpy PROTO_LIST ((POINTER, POINTER, unsigned int));
static void MD4_memset PROTO_LIST ((POINTER, int, unsigned int));

static unsigned char PADDING[64] = {
  0x80, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
};

/* F, G and H are basic MD4 functions.
 */
#define F(x, y, z) (((x) & (y)) | ((~x) & (z)))
#define G(x, y, z) (((x) & (y)) | ((x) & (z)) | ((y) & (z)))
#define H(x, y, z) ((x) ^ (y) ^ (z))

/* ROTATE_LEFT rotates x left n bits.
 */
#define ROTATE_LEFT(x, n) (((x) << (n)) | ((x) >> (32-(n))))

/* FF, GG and HH are transformations for rounds 1, 2 and 3 */
/* Rotation is separate from addition to prevent recomputation */
```

```
#define FF(a, b, c, d, x, s) { \
    (a) += F ((b), (c), (d)) + (x); \
    (a) = ROTATE_LEFT ((a), (s)); \
  }
#define GG(a, b, c, d, x, s) { \
    (a) += G ((b), (c), (d)) + (x) + (UINT4)0x5a827999; \
    (a) = ROTATE_LEFT ((a), (s)); \
  }
#define HH(a, b, c, d, x, s) { \
    (a) += H ((b), (c), (d)) + (x) + (UINT4)0x6ed9eba1; \
    (a) = ROTATE_LEFT ((a), (s)); \
  }

/* MD4 initialization. Begins an MD4 operation, writing a new context.
 */
void MD4Init (context)
MD4_CTX *context;                                        /* context */
{
  context->count[0] = context->count[1] = 0;

  /* Load magic initialization constants.
   */
  context->state[0] = 0x67452301;
  context->state[1] = 0xefcdab89;
  context->state[2] = 0x98badcfe;
  context->state[3] = 0x10325476;
}

/* MD4 block update operation. Continues an MD4 message-digest
     operation, processing another message block, and updating the
     context.
 */
void MD4Update (context, input, inputLen)
MD4_CTX *context;                                        /* context */
unsigned char *input;                                /* input block */
unsigned int inputLen;                     /* length of input block */
{
  unsigned int i, index, partLen;

  /* Compute number of bytes mod 64 */
  index = (unsigned int)((context->count[0] >> 3) & 0x3F);
  /* Update number of bits */
  if ((context->count[0] += ((UINT4)inputLen << 3))
      < ((UINT4)inputLen << 3))
    context->count[1]++;
  context->count[1] += ((UINT4)inputLen >> 29);

  partLen = 64 - index;
```

```
  /* Transform as many times as possible.
   */
  if (inputLen >= partLen) {
    MD4_memcpy
      ((POINTER)&context->buffer[index], (POINTER)input, partLen);
    MD4Transform (context->state, context->buffer);

    for (i = partLen; i + 63 < inputLen; i += 64)
      MD4Transform (context->state, &input[i]);

    index = 0;
  }
  else
    i = 0;

  /* Buffer remaining input */
  MD4_memcpy
    ((POINTER)&context->buffer[index], (POINTER)&input[i],
     inputLen-i);
}

/* MD4 finalization. Ends an MD4 message-digest operation, writing the
     the message digest and zeroizing the context.
 */
void MD4Final (digest, context)
unsigned char digest[16];                         /* message digest */
MD4_CTX *context;                                       /* context */
{
  unsigned char bits[8];
  unsigned int index, padLen;

  /* Save number of bits */
  Encode (bits, context->count, 8);

  /* Pad out to 56 mod 64.
   */
  index = (unsigned int)((context->count[0] >> 3) & 0x3f);
  padLen = (index < 56) ? (56 - index) : (120 - index);
  MD4Update (context, PADDING, padLen);

  /* Append length (before padding) */
  MD4Update (context, bits, 8);
  /* Store state in digest */
  Encode (digest, context->state, 16);

  /* Zeroize sensitive information.
   */
  MD4_memset ((POINTER)context, 0, sizeof (*context));
```

```
}

/* MD4 basic transformation. Transforms state based on block.
 */
static void MD4Transform (state, block)
UINT4 state[4];
unsigned char block[64];
{
  UINT4 a = state[0], b = state[1], c = state[2], d = state[3], x[16];

  Decode (x, block, 64);

  /* Round 1 */
  FF (a, b, c, d, x[ 0], S11); /* 1 */
  FF (d, a, b, c, x[ 1], S12); /* 2 */
  FF (c, d, a, b, x[ 2], S13); /* 3 */
  FF (b, c, d, a, x[ 3], S14); /* 4 */
  FF (a, b, c, d, x[ 4], S11); /* 5 */
  FF (d, a, b, c, x[ 5], S12); /* 6 */
  FF (c, d, a, b, x[ 6], S13); /* 7 */
  FF (b, c, d, a, x[ 7], S14); /* 8 */
  FF (a, b, c, d, x[ 8], S11); /* 9 */
  FF (d, a, b, c, x[ 9], S12); /* 10 */
  FF (c, d, a, b, x[10], S13); /* 11 */
  FF (b, c, d, a, x[11], S14); /* 12 */
  FF (a, b, c, d, x[12], S11); /* 13 */
  FF (d, a, b, c, x[13], S12); /* 14 */
  FF (c, d, a, b, x[14], S13); /* 15 */
  FF (b, c, d, a, x[15], S14); /* 16 */

  /* Round 2 */
  GG (a, b, c, d, x[ 0], S21); /* 17 */
  GG (d, a, b, c, x[ 4], S22); /* 18 */
  GG (c, d, a, b, x[ 8], S23); /* 19 */
  GG (b, c, d, a, x[12], S24); /* 20 */
  GG (a, b, c, d, x[ 1], S21); /* 21 */
  GG (d, a, b, c, x[ 5], S22); /* 22 */
  GG (c, d, a, b, x[ 9], S23); /* 23 */
  GG (b, c, d, a, x[13], S24); /* 24 */
  GG (a, b, c, d, x[ 2], S21); /* 25 */
  GG (d, a, b, c, x[ 6], S22); /* 26 */
  GG (c, d, a, b, x[10], S23); /* 27 */
  GG (b, c, d, a, x[14], S24); /* 28 */
  GG (a, b, c, d, x[ 3], S21); /* 29 */
  GG (d, a, b, c, x[ 7], S22); /* 30 */
  GG (c, d, a, b, x[11], S23); /* 31 */
  GG (b, c, d, a, x[15], S24); /* 32 */
```

```
  /* Round 3 */
  HH (a, b, c, d, x[ 0], S31); /* 33 */
  HH (d, a, b, c, x[ 8], S32); /* 34 */
  HH (c, d, a, b, x[ 4], S33); /* 35 */
  HH (b, c, d, a, x[12], S34); /* 36 */
  HH (a, b, c, d, x[ 2], S31); /* 37 */
  HH (d, a, b, c, x[10], S32); /* 38 */
  HH (c, d, a, b, x[ 6], S33); /* 39 */
  HH (b, c, d, a, x[14], S34); /* 40 */
  HH (a, b, c, d, x[ 1], S31); /* 41 */
  HH (d, a, b, c, x[ 9], S32); /* 42 */
  HH (c, d, a, b, x[ 5], S33); /* 43 */
  HH (b, c, d, a, x[13], S34); /* 44 */
  HH (a, b, c, d, x[ 3], S31); /* 45 */
  HH (d, a, b, c, x[11], S32); /* 46 */
  HH (c, d, a, b, x[ 7], S33); /* 47 */
  HH (b, c, d, a, x[15], S34); /* 48 */

  state[0] += a;
  state[1] += b;
  state[2] += c;
  state[3] += d;

  /* Zeroize sensitive information.
   */
  MD4_memset ((POINTER)x, 0, sizeof (x));
}

/* Encodes input (UINT4) into output (unsigned char). Assumes len is
     a multiple of 4.
 */
static void Encode (output, input, len)
unsigned char *output;
UINT4 *input;
unsigned int len;
{
  unsigned int i, j;

  for (i = 0, j = 0; j < len; i++, j += 4) {
    output[j] = (unsigned char)(input[i] & 0xff);
    output[j+1] = (unsigned char)((input[i] >> 8) & 0xff);
    output[j+2] = (unsigned char)((input[i] >> 16) & 0xff);
    output[j+3] = (unsigned char)((input[i] >> 24) & 0xff);
  }
}

/* Decodes input (unsigned char) into output (UINT4). Assumes len is
     a multiple of 4.
```

```
 */
static void Decode (output, input, len)

UINT4 *output;
unsigned char *input;
unsigned int len;
{
  unsigned int i, j;

  for (i = 0, j = 0; j < len; i++, j += 4)
    output[i] = ((UINT4)input[j]) | (((UINT4)input[j+1]) << 8) |
      (((UINT4)input[j+2]) << 16) | (((UINT4)input[j+3]) << 24);
}

/* Note: Replace "for loop" with standard memcpy if possible.
 */
static void MD4_memcpy (output, input, len)
POINTER output;
POINTER input;
unsigned int len;
{
  unsigned int i;

  for (i = 0; i < len; i++)
    output[i] = input[i];
}

/* Note: Replace "for loop" with standard memset if possible.
 */
static void MD4_memset (output, value, len)
POINTER output;
int value;
unsigned int len;
{
  unsigned int i;

  for (i = 0; i < len; i++)
    ((char *)output)[i] = (char)value;
}
```

A.4 mddriver.c

```
/* MDDRIVER.C - test driver for MD2, MD4 and MD5
 */

/* Copyright (C) 1990-2, RSA Data Security, Inc. Created 1990. All
   rights reserved.
```

```
   RSA Data Security, Inc. makes no representations concerning either
   the merchantability of this software or the suitability of this
   software for any particular purpose. It is provided "as is"
   without express or implied warranty of any kind.

   These notices must be retained in any copies of any part of this
   documentation and/or software.

 */

/* The following makes MD default to MD5 if it has not already been
     defined with C compiler flags.
 */
#ifndef MD
#define MD MD5
#endif

#include <stdio.h>
#include <time.h>
#include <string.h>
#include "global.h"
#if MD == 2
#include "md2.h"
#endif
#if MD == 4
#include "md4.h"
#endif
#if MD == 5
#include "md5.h"
#endif

/* Length of test block, number of test blocks.
 */
#define TEST_BLOCK_LEN 1000
#define TEST_BLOCK_COUNT 1000

static void MDString PROTO_LIST ((char *));
static void MDTimeTrial PROTO_LIST ((void));
static void MDTestSuite PROTO_LIST ((void));
static void MDFile PROTO_LIST ((char *));
static void MDFilter PROTO_LIST ((void));
static void MDPrint PROTO_LIST ((unsigned char [16]));

#if MD == 2
#define MD_CTX MD2_CTX
#define MDInit MD2Init
#define MDUpdate MD2Update
#define MDFinal MD2Final
```

```
#endif
#if MD == 4
#define MD_CTX MD4_CTX
#define MDInit MD4Init
#define MDUpdate MD4Update
#define MDFinal MD4Final
#endif
#if MD == 5
#define MD_CTX MD5_CTX
#define MDInit MD5Init
#define MDUpdate MD5Update
#define MDFinal MD5Final
#endif

/* Main driver.

   Arguments (may be any combination):
     -sstring - digests string
     -t       - runs time trial
     -x       - runs test script
     filename - digests file
     (none)   - digests standard input
 */
int main (argc, argv)
int argc;
char *argv[];
{
  int i;

  if (argc > 1)
    for (i = 1; i < argc; i++)
      if (argv[i][0] == '-' && argv[i][1] == 's')
        MDString (argv[i] + 2);
      else if (strcmp (argv[i], "-t") == 0)
        MDTimeTrial ();
      else if (strcmp (argv[i], "-x") == 0)
        MDTestSuite ();
      else
        MDFile (argv[i]);
  else
    MDFilter ();

  return (0);
}

/* Digests a string and prints the result.
 */
static void MDString (string)
```

```
char *string;
{
  MD_CTX context;
  unsigned char digest[16];
  unsigned int len = strlen (string);

  MDInit (&context);
  MDUpdate (&context, string, len);
  MDFinal (digest, &context);

  printf ("MD%d (\"%s\") = ", MD, string);
  MDPrint (digest);
  printf ("\n");
}

/* Measures the time to digest TEST_BLOCK_COUNT TEST_BLOCK_LEN-byte
     blocks.
 */
static void MDTimeTrial ()
{
  MD_CTX context;
  time_t endTime, startTime;
  unsigned char block[TEST_BLOCK_LEN], digest[16];
  unsigned int i;

  printf
    ("MD%d time trial. Digesting %d %d-byte blocks ...", MD,
     TEST_BLOCK_LEN, TEST_BLOCK_COUNT);

  /* Initialize block */
  for (i = 0; i < TEST_BLOCK_LEN; i++)
    block[i] = (unsigned char)(i & 0xff);

  /* Start timer */
  time (&startTime);

  /* Digest blocks */
  MDInit (&context);
  for (i = 0; i < TEST_BLOCK_COUNT; i++)
    MDUpdate (&context, block, TEST_BLOCK_LEN);
  MDFinal (digest, &context);

  /* Stop timer */
  time (&endTime);

  printf (" done\n");
  printf ("Digest = ");
  MDPrint (digest);
```

```
  printf ("\nTime = %ld seconds\n", (long)(endTime-startTime));
  printf
    ("Speed = %ld bytes/second\n",
     (long)TEST_BLOCK_LEN * (long)TEST_BLOCK_COUNT/(endTime-startTime));
}

/* Digests a reference suite of strings and prints the results.
 */
static void MDTestSuite ()
{
  printf ("MD%d test suite:\n", MD);

  MDString ("");
  MDString ("a");
  MDString ("abc");
  MDString ("message digest");
  MDString ("abcdefghijklmnopqrstuvwxyz");
  MDString
    ("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789");
  MDString

    ("12345678901234567890123456789012345678901\
2345678901234567890123456789012345678901");
}

/* Digests a file and prints the result.
 */
static void MDFile (filename)
char *filename;
{
  FILE *file;
  MD_CTX context;
  int len;
  unsigned char buffer[1024], digest[16];

  if ((file = fopen (filename, "rb")) == NULL)
    printf ("%s can't be opened\n", filename);

  else {
    MDInit (&context);
    while (len = fread (buffer, 1, 1024, file))
      MDUpdate (&context, buffer, len);
    MDFinal (digest, &context);

    fclose (file);

    printf ("MD%d (%s) = ", MD, filename);
    MDPrint (digest);
```

```
    printf ("\n");
  }
}

/* Digests the standard input and prints the result.
 */
static void MDFilter ()
{
  MD_CTX context;
  int len;
  unsigned char buffer[16], digest[16];

  MDInit (&context);
  while (len = fread (buffer, 1, 16, stdin))
    MDUpdate (&context, buffer, len);
  MDFinal (digest, &context);

  MDPrint (digest);
  printf ("\n");
}

/* Prints a message digest in hexadecimal.
 */
static void MDPrint (digest)
unsigned char digest[16];

{
  unsigned int i;

  for (i = 0; i < 16; i++)
    printf ("%02x", digest[i]);
}
```

A.5 Test suite

The MD4 test suite (driver option "-x") should print the following results:

```
MD4 test suite:
MD4 ("") = 31d6cfe0d16ae931b73c59d7e0c089c0
MD4 ("a") = bde52cb31de33e46245e05fbdbd6fb24
MD4 ("abc") = a448017aaf21d8525fc10ae87aa6729d
MD4 ("message digest") = d9130a8164549fe818874806e1c7014b
MD4 ("abcdefghijklmnopqrstuvwxyz") = d79e1c308aa5bbcdeea8ed63df412da9
MD4 ("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789") =
043f8582f241db351ce627e153e7f0e4
MD4 ("12345678901234567890123456789012345678901234567890123456789012345678
78901234567890") = e33b4ddc9c38f2199c3e7b164fcc0536
```

Security Considerations

The level of security discussed in this memo is considered to be sufficient for implementing moderate security hybrid digital-signature schemes based on MD4 and a public-key cryptosystem. We do not know of any reason that MD4 would not be sufficient for implementing very high security digital-signature schemes, but because MD4 was designed to be exceptionally fast, it is "at the edge" in terms of risking successful cryptanalytic attack. After further critical review, it may be appropriate to consider MD4 for very high security applications. For very high security applications before the completion of that review, the MD5 algorithm [4] is recommended.

Author's Address

Ronald L. Rivest
Massachusetts Institute of Technology
Laboratory for Computer Science
NE43-324
545 Technology Square
Cambridge, MA 02139-1986

Phone: (617) 253-5880
EMail: rivest@theory.lcs.mit.edu